CSE216 Programming Abstractions
Data Abstractions

YoungMin Kwon

Overview: 3 Elements of Data

- The 3 elements of data
  - Primitive data
  - Compound data
  - Data abstraction

  - Like 3 elements of programming
    - Primitive expression
    - Means of combination
    - Means of abstraction

Overview: Primitive Data

- Integers
  - -1, 0, 1, 2, ...
- Floats
  - -1.0, 0, 1.0, 3.141592, ...
- Boolean
  - true, false
- Character
  - 'a', 'b', 'c',
- String
  - "hello world"

```
# 1;;
- : int = 1

# 1.0;;
- : float = 1.

# true;;
-  : bool = true

# 'a';;
- : char = 'a'

# "hello";;
- : string = "hello"
```

Overview: Compound Data

- Compound data
  - A way to glue data together
    - Closure property: can glue combined data objects again
  - Needs a way to access individual components
  - Compound data can increase the modularity of programs

Overview: Compound Data

- E.g.) Rational number with two integers
  - Without compound data: needs to manage sets of two integer variables

```
# let num1 = 1 in let den1 = 2 in
  let num2 = 3 in let den2 = 4 in
  let num3 = add_rat_num(num1, den1, num2, den2) in
  let den3 = add_rat_den(num1, den1, num2, den2) in …
```

  - Combine num and den into rat

```
# let rat1 = make_rat(1,2) in
  let rat2 = make_rat(3,4) in
  let rat3 = add_rat(rat1, rat2) in …
```

Overview: Data Abstraction

- Data abstraction means isolating
  - how data objects are represented from
  - how data objects are used
  - E.g.)

```
let example () =
    let ( + ) = arith "add" in
    let ( - ) = arith "sub" in
    let ( * ) = arith "mul" in
    let ( / ) = arith "div" in
    let a = complex 2. 3. in
    let b = polar   1. 3.14 in
    (a + b) * a / b
```

  - a is a complex number in the rectangular form
  - b is a complex number in the polar form
  - However, we can use them the same way without distinguishing their implementations

Primitive Data

- OCaml Basic types

Type	Comments
int	31-bit signed int on 32-bit processors, 63-bit signed int on 64-bit processors
float	IEEE double-precision floating point
bool	A boolean
char	An 8-bit char
string	A string
unit	Like void in C

Compound Data: Tuples

- Tuple
  - Ordered collection of values that can be of different type
  - E.g.)

```
# (1, "hello", true);;
-  : int * string * bool = (1, "hello", true)

# (1, ("hello", true));;
-  : int * (string * bool) = (1, ("hello", true))
```

Compound Data: Tuples

- Pattern matching to access components

```
# let (x, y) = (1, ("hello", true));;
val x : int = 1
val y : string * bool = ("hello", true)
```

```
# let (x, (y, z)) = (1, ("hello", true));;
val x : int = 1
val y : string = "hello"
val z : bool = true
```

```
# let (_, (y, _)) = (1, ("hello", true));;
val y : string = "hello"
```

Building Rational Numbers

- Example: building rational numbers
  - Assume that the constructor and selectors are available as
    - make_rat n d,
    - num x, den x

```
let add_rat x y =
    make_rat ((num x) * (den y) + (num y) * (den x))
             ((den x) * (den y));;

let sub_rat x y =
    make_rat ((num x) * (den y) - (num y) * (den x))
             ((den x) * (den y));;
```

Building Rational Numbers

```
let mul_rat x y =
    make_rat ((num x) * (num y)) ((den x) * (den y));;


let div_rat x y =
    make_rat ((num x) * (den y)) ((den x) * (num y));;


let equal_rat x y =
    (num x) * (den y) = (den x) * (num y);;

let print_rat x =
    Printf.printf "%d/%d\n" (num x) (den x);;
```

Building Rational Numbers

- Representing rational numbers as a pair
  - Implementing pair using a tuple: constructor and accessors

```
let pair a b = (a, b);;
let first  x = let (a, _) = x in a;;
let second x = let (_, b) = x in b;;
```

  - The constructor and accessors for rational numbers

```
let make_rat n d = pair n d;;
let num x = first x;;
let den x = second x;;

print_rat (sub_rat (make_rat 1 2)
                   (make_rat 1 3));;
```

Building Rational Numbers

- Reduce rational numbers to their lowest terms
  - Divide n and d by their gcd in make_rat

```
let make_rat n d =
    let rec gcd x y =
        if x > y      then gcd (x - y) y
        else if x < y then gcd (y - x) x
        else x in
    let g = gcd n d in
    pair (n/g) (d/g);;

print_rat (sub_rat (make_rat 1 2)
                   (make_rat 1 3));;
```

  - Because of the data abstraction, this change does not affect other parts of the program

Building Rational Numbers

- Implementing pair using a function

```
let pair a b = fun z -> if z then a else b;;
let first  x = x true in;;
let second x = x false in;;

print_rat (sub_rat (make_rat 1 2)
                   (make_rat 1 3));;
```

- Again, because of the data abstraction, this change does not affect any other parts of the program

What is Meant by Data

- We can think of data as
  - Some collection of selectors and constructors, and
  - Conditions that these procedures must satisfy

  - E.g.) pairs of rational number
    - Constructor: pair
    - Selectors: first, second
    - Conditions: if *x* is a *pair* of *a* and *b*, then *first x* is *a* and *second x* is *b*

What is Meant by Data

- E.g.) Representing pair

```
let pair a b = (a, b)
let first  (a, _) = a
let second (_, b) = b
```

uncurried function: pattern matching at parameters

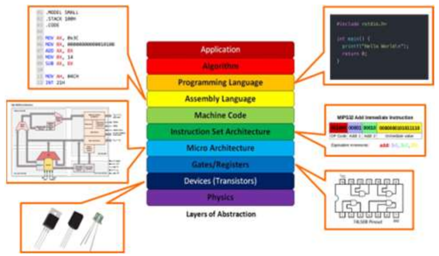
```
let pair a b z =
    if z then a
         else b
let first  x = x true
let second x = x false
```

- Both representations have the same constructor, selectors, and the condition

Abstraction Barriers

- Abstraction barriers
  - Isolate different levels of a system
  - The barrier at each level
    - Separates the program above that uses the data
    - From the program below that implements the data abstraction
  - Procedures at each level are interfaces that define the abstraction barriers

Abstraction Barriers



Programs that use rational numbers

Rational numbers in problem domain

add_rat sub_rat …

Rational numbers as numerators and denominators

make_rat num den

Rational numbers as pairs

pair first second

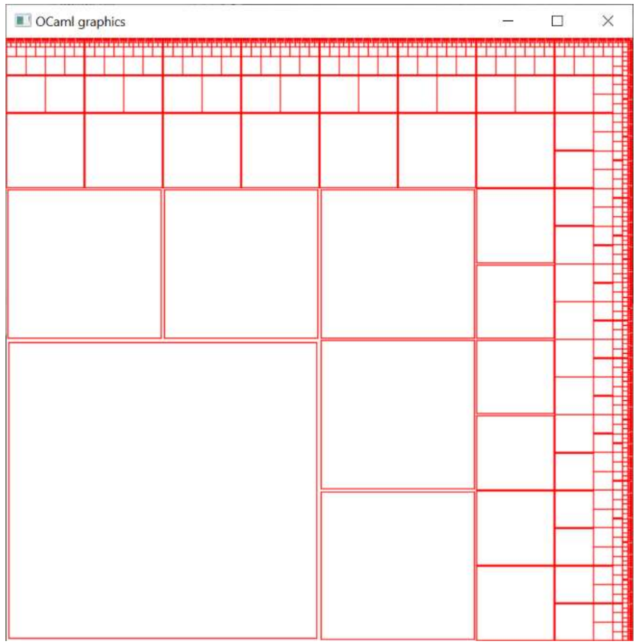
Pairs as tuples

(a, b) let (a, b) = x

However tuples are implemented

Example: A Picture Language

- Demonstrates the power of
  - Data abstraction
  - High order procedures
  - Closure property
    - Results of an operation can be used for the same operation



Install Graphics Package

- Run the following commands in Ubuntu
  - sudo apt install pkg-config (may not necessary)
  - opam init
  - opam update
  - opam install graphics

Install Graphics Package

- Copy graphics.cmi and graphics.cma to your local directory
  - opam config list graphics
  - Find where the graphics library is installed
    - Look for graphics:lib or library directory for this package
  - Copy graphics.cmi and graphics.cma from the library to your local directory
    - E.g.:
    - cp ~/.opam/default/lib/graphics/graphics.cmi .
    - cp ~/.opam/default/lib/graphics/graphics.cma .

Test Graphics

- Run the following commands from your ocaml top level

```
ykwon4@youngbox2: /mnt/c,
ykwon4@youngbox2:/mnt/c/Users/young/Documents/Share/CSE216/OCaml/Recitation$ ocaml
        OCaml version 4.13.1

# #load "graphics.cma";;
# open Graphics;;
# open_graph " 500x500";;
- : unit = ()
# lineto 300 300;;
- : unit = ()
# close_graph ();;
- : unit = ()
# 
```

Install X11 Server

- You may need to install X11 server
  - Windows: install xming from https://sourceforge.net/projects/xming/
    - WSL: may need to add export DISPLAY=127.0.0.1:0 to .bashrc file
  - Mac: install XQuartz

To Use Graphics in Cygwin

- Check if Graphics package is installed

```
$ opam list
# Packages matching: installed
# Name              # Installed      # Synopsis
base-bigarray       base
...
graphics            5.1.1            The OCaml graphics library
ocaml               4.11.1           The OCaml compiler (virtual package)
...
```

- Install Graphics package if it is not installed

```
$ opam install graphics
```

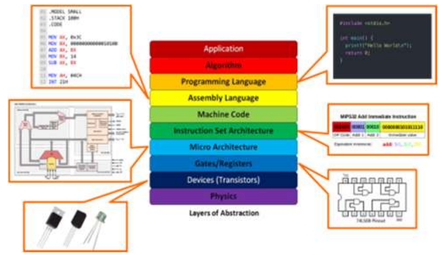
- Run Ocaml with -I (include) option

```
$ ocaml -I $(ocamlfind query graphics)
```

- If ocamlfind is not installed, install it using

```
$ opam install ocamlfind
```

Picture Language: Abstraction Barriers



Programs that use transforms

Complex transform operations on painter

right_split, up_split, corner_split…

Simple transform operations on painter

tf_painter, flip, scale, translate, rotate

Frames as a tuple of vectors

new_frame, frame_to_globalcoord_map

2D vectors as tuples

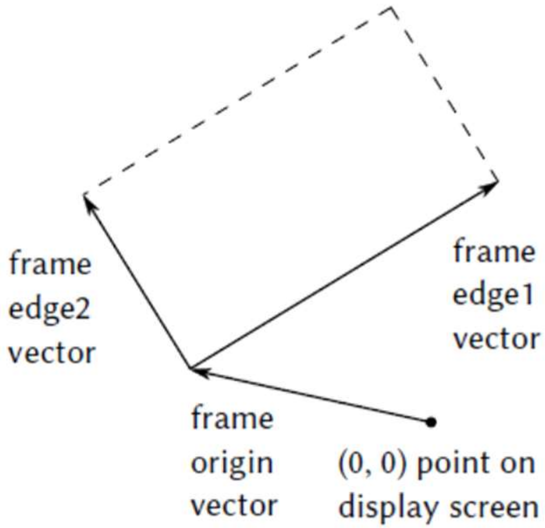
add, sub, prod, smul

However tuples are implemented

A Picture Language

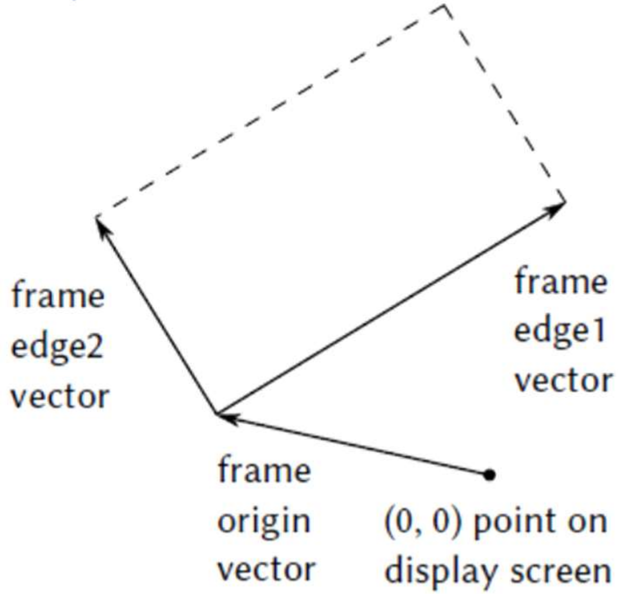
- Key elements
  - Painter
    - A function that takes a frame and draws on the frame

  - Frame
    - Decides where and how the painter draws image
    - A tuple of o, u, and v vectors in the global coordinate
      - o: origin vector,
      - u: edge1 vector, v: edge2 vector

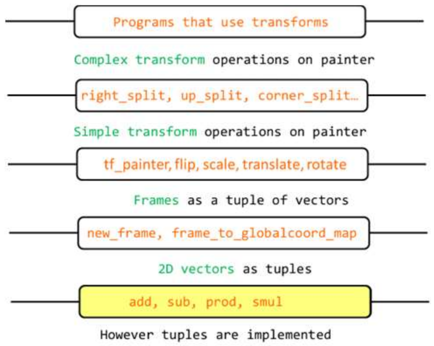


A Picture Language

- Key elements
  - Mapping
    - Frame coordinate → screen coordinate
    - p → o + p.x * u + p.y * v
    - Painter draws on the frame
    - We transform the frames

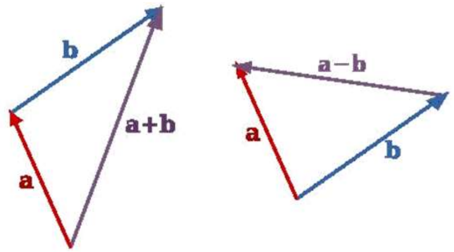


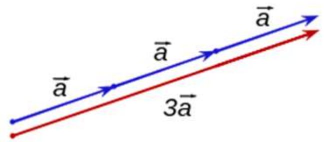
A Picture Language

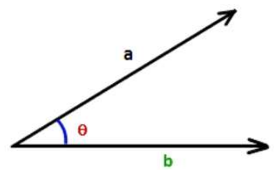


- Vector 2d

```
(*vector 2d-------------------------------------
*)

(*add, sub*)
let add  (x1, y1) (x2, y2) = (x1 +. x2, y1 +. y2)
let sub  (x1, y1) (x2, y2) = (x1 -. x2, y1 -. y2)

(*scalar multiplication*)
let smul s (x, y) = (s *. x, s *. y)

(*inner product*)
let prod (x1, y1) (x2, y2) = x1 *. x2 +. y1 *. y2
```



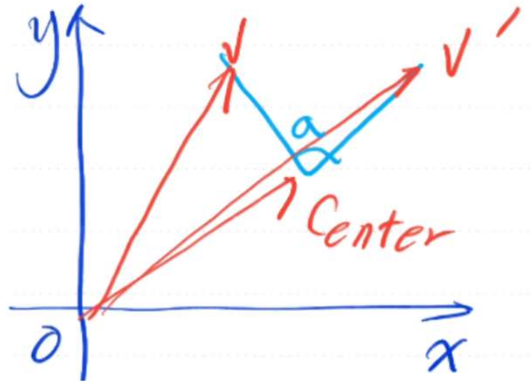




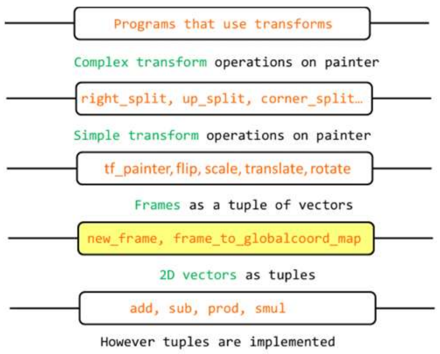
$a.b = |a|\ |b|\ \cos\theta$

A Picture Language

- Vector 2d

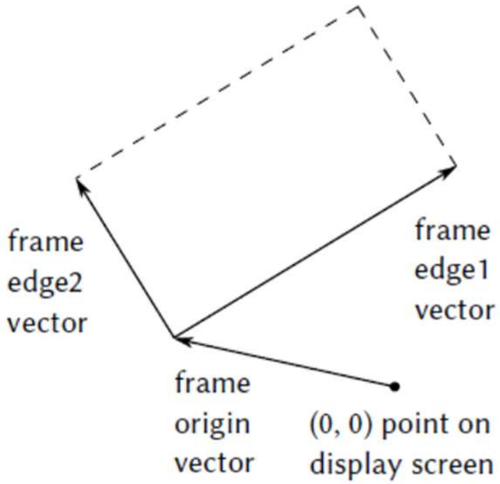



```
let pi = acos (- 1.)
let deg2rad deg = deg /. 180. *. pi
let rad2deg rad = rad /. pi *. 180.

(*rotate v a degree from center*)
let rot a center v =
    let cv = sub v center in
    let cosx = cos (deg2rad a) in
    let sinx = sin (deg2rad a) in
    let x = prod (cosx, -. sinx) cv in
    let y = prod (sinx,    cosx) cv in
    add (x, y) center
```



Frame and Coordinate Mapping

```
(*frame----------------------------------------
 *)

let new_frame o u v = (o, u, v)
let frame_g = new_frame (0.,0.) (1.,0.) (0.,1.)

(*convert (x,y) in frame coord to global coord*)
let frame_to_global_coord_map frame =
    let (o, u, v) = frame in
    fun (x, y) -> add o (add (smul x u) (smul y v))
```



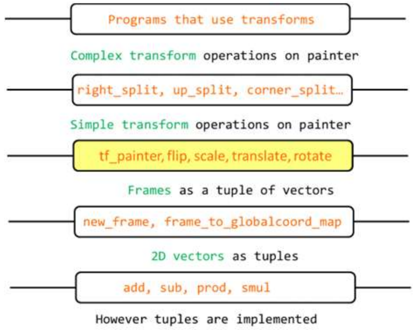
Base Painter

```
(*base painter--------------------------------------
  draw a box of a nearly entire frame
*)
let base_painter =
    let scale a s = truncate (a *. float s) in
    let move_to (x, y) = scale x (size_x ()) |> fun sx ->
                         scale y (size_y ()) |> fun sy ->
                         moveto sx sy in
    let line_to (x, y) = scale x (size_x ()) |> fun sx ->
                         scale y (size_y ()) |> fun sy ->
                         lineto sx sy in
    fun frame ->
        let map = frame_to_global_coord_map frame in
        let b = 0.99 in
        let a = 1. -. b in
        set_color red;
        move_to (map (a, a));
        line_to (map (a, b));
        line_to (map (b, b));
        line_to (map (b, a));
        line_to (map (a, a))
```

Returns a painter, a function that takes a frame and draws on it

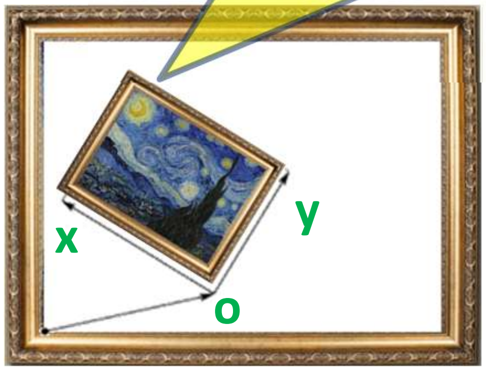
Sequence Operator: append next expr if prev expr is ()

Simple Transform Painters



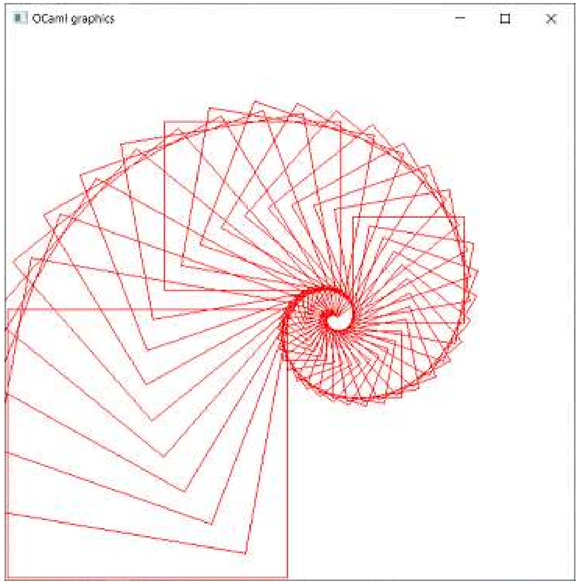

```
(*simple transform on painters----------------
 *)
(*tf_painter make painter draw on the local
  coordinate system of o, x, y w.r.t. frame
  i.e. paint on the new frame of o, x, y w.r.t. frame*)

let tf_painter painter o x y =
    fun frame ->
        let map = frame_to_global_coord_map frame in
        let (go, gu, gv) = (map o, map x, map y) in

        (*make the frame for o, x, y local coord. sys.*)
        painter (new_frame go (sub gu go) (sub gv go))
```

Closure property: tf_painter returns a painter. It takes a frame and draws on it

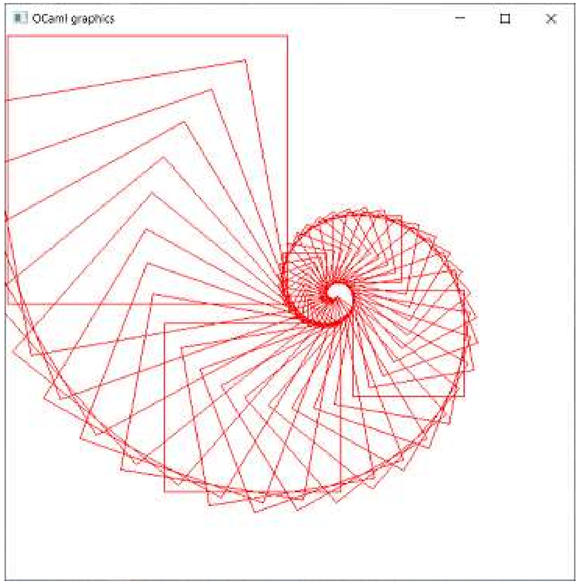


Simple Transform Painters

```
let flip_ver painter =
    tf_painter painter (0.,1.) (1.,1.) (0.,0.)
    (*                 ^o      ^x      ^y   *)
```

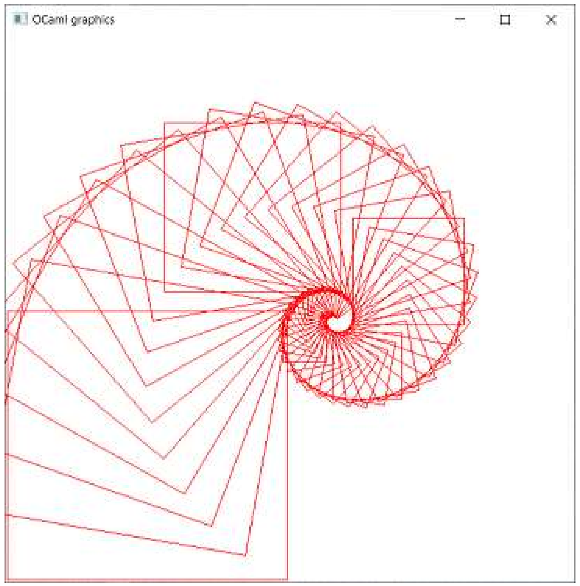


=>

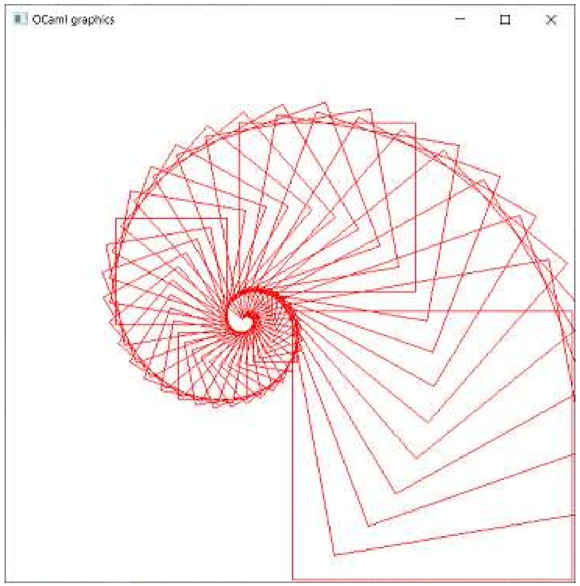


Simple Transform Painters

```
let flip_hor painter =
    tf_painter painter (1.,0.) (0.,0.) (1.,1.)
    (*                 ^o      ^x      ^y   *)
```



=>



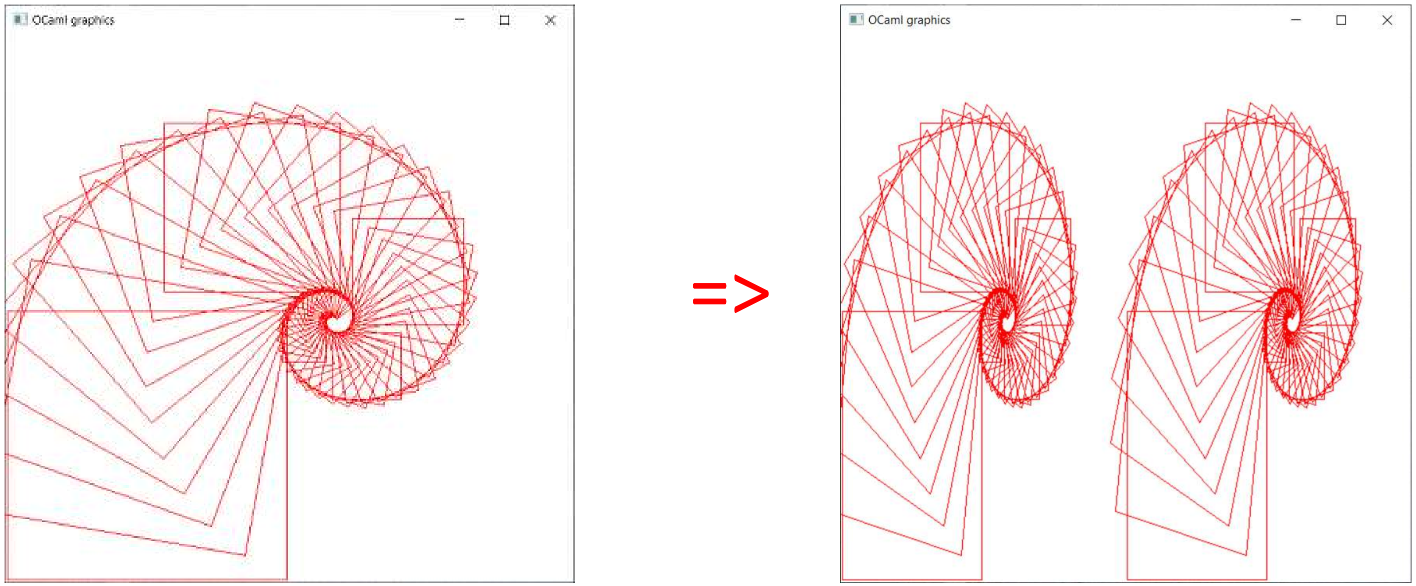
Simple Transform Painters

```
let scale sx sy painter =
    tf_painter painter (0., 0.) (sx, 0.) (0., sy)

let translate tx ty painter =
    tf_painter painter (tx, ty) (1. +. tx, 0. +. ty)
                                (0. +. tx, 1. +. ty)

let rotate a center painter =
    let r = rot a center in
    tf_painter painter (r (0., 0.)) (r (1., 0.)) (r (0., 1.))

let rotate90  painter = rotate 90.  (0.5, 0.5) painter
let rotate180 painter = rotate 180. (0.5, 0.5) painter
let rotate270 painter = rotate 270. (0.5, 0.5) painter
```

Simple Transform Painters

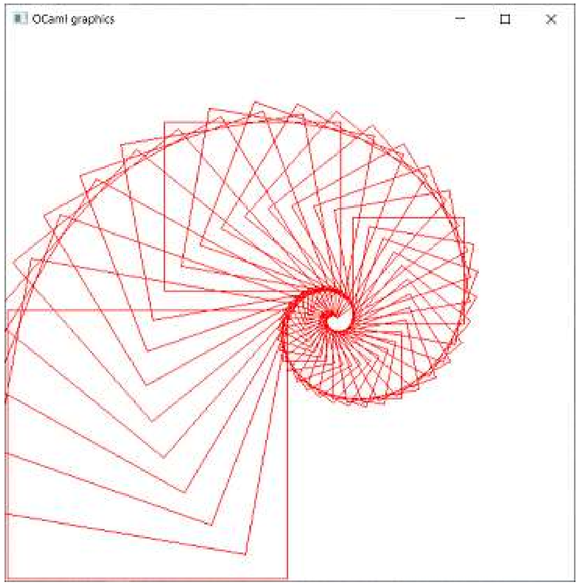
```
let beside painter_l painter_r =
    let paint_left  = tf_painter painter_l (0.,0.)  (0.5,0.) (0.,1.)  in
    let paint_right = tf_painter painter_r (0.5,0.) (1.,0.)  (0.5,1.) in
    fun frame ->
        paint_left  frame;
        paint_right frame
```

Closure property: beside returns a painter. It takes a frame and draws on it

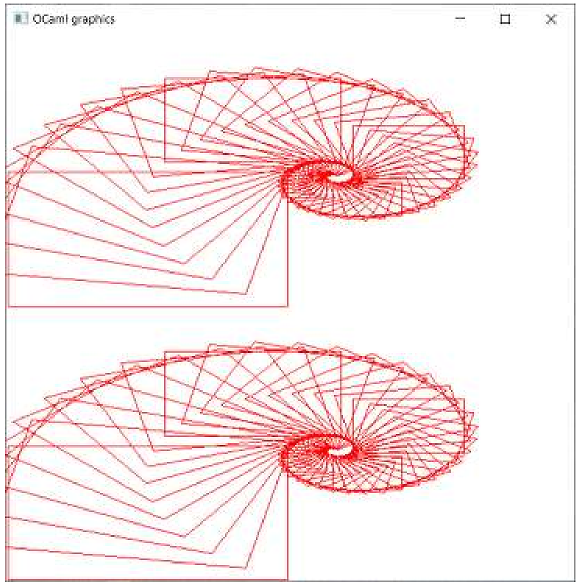


Simple Transform Painters

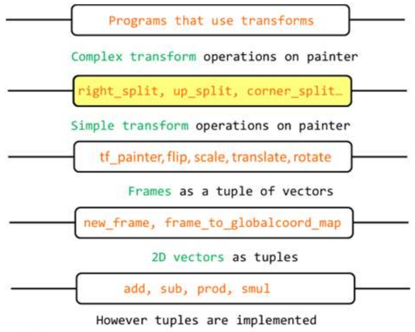
```
let below painter_t painter_b =
    let paint_top    = tf_painter painter_t (0.,0.5) (1.,0.5) (0.,1.)  in
    let paint_bottom = tf_painter painter_b (0.,0.)  (1.,0.)  (0.,0.5) in
    fun frame ->
        paint_top    frame;
        paint_bottom frame
```



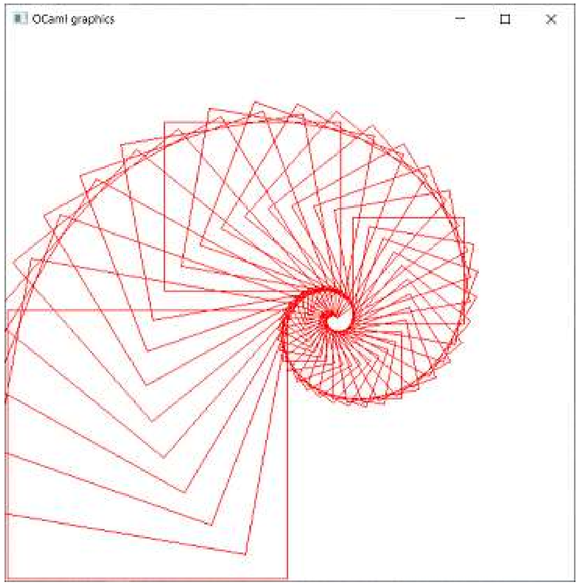
=>



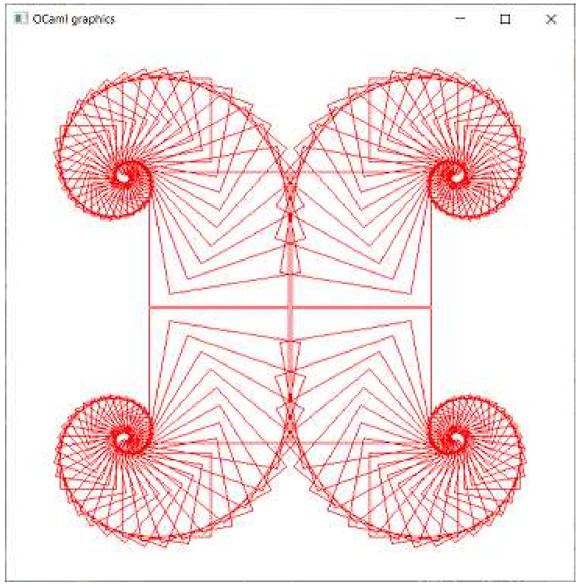
Complex Transform Painters




```
(*complex transform on painters-------------------
*)
let flipped_pairs painter =
    let painter2 = beside (flip_hor painter) painter in
    below painter2 (flip_ver painter2)
```



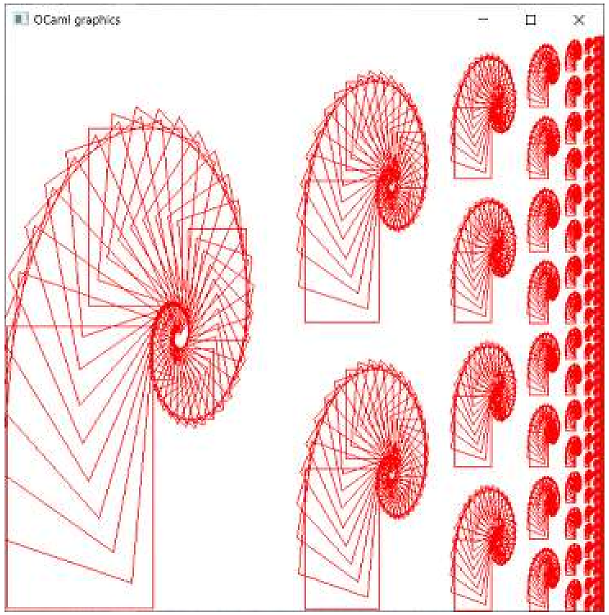
=>



Complex Transform Painters

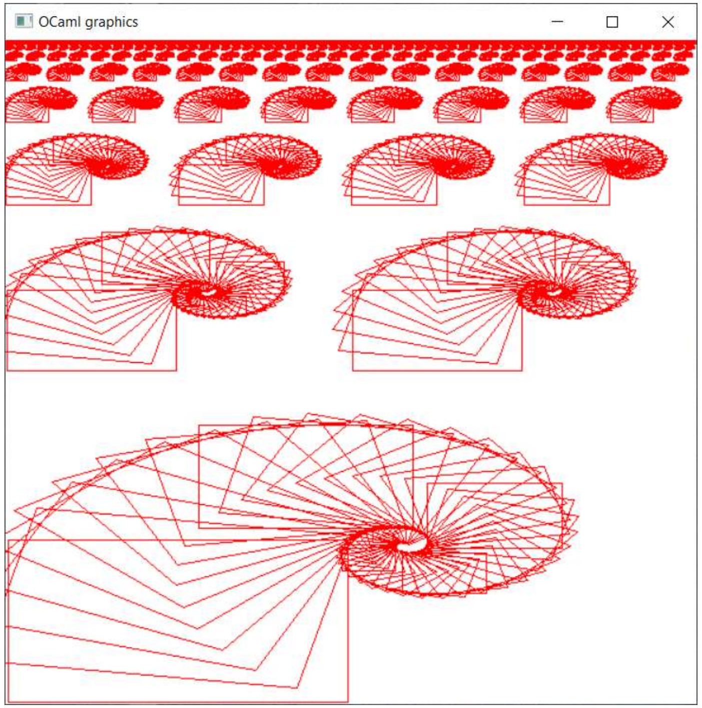
```
let rec right_split painter n =
    if n = 0 then painter
    else
        let smaller = right_split painter (n-1) in
        beside painter (below smaller smaller)
```

right_split returns a painter: it takes a frame and draws on it



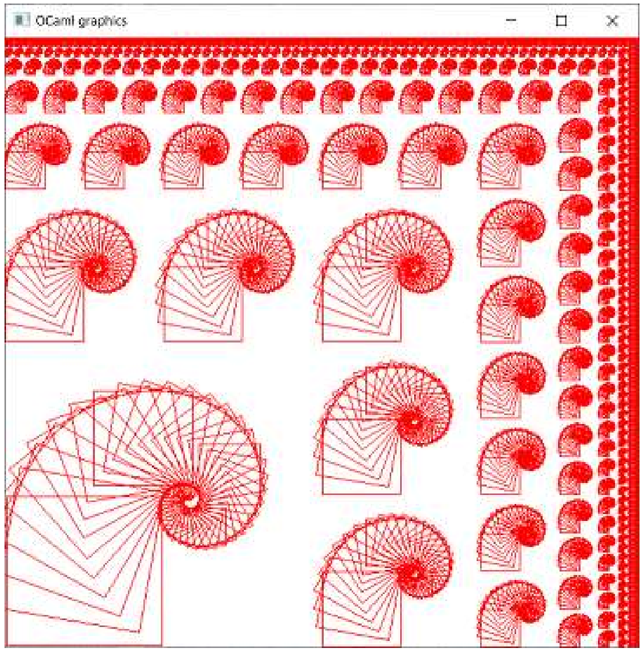
Complex Transform Painters

```
let rec up_split painter n =
    if n = 0 then painter
    else
        let smaller = up_split painter (n-1) in
        below (beside smaller smaller) painter
```



Complex transform on painter

```
let rec corner_split painter n =
    if n = 0 then painter
    else
        let up           = up_split     painter (n-1) in
        let right        = right_split  painter (n-1) in
        let top_left     = beside       up up         in
        let bottom_right = below        right right   in
        let corner       = corner_split painter (n-1) in
        beside (below top_left painter)
               (below corner bottom_right)
```

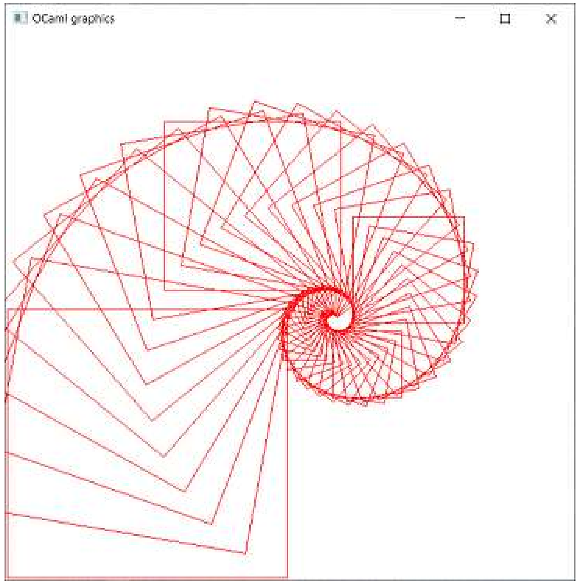


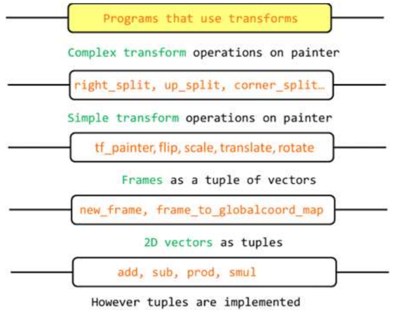
Without up up or right right, the pictures look squeezed.

Complex Transform Painters

```
let rec rot_scale painter n =
    if n = 0 then painter
    else
        let rs = painter |> scale 0.95 0.95
                         |> rotate (-10.) (0.7, 0.3)
                         |> fun p -> rot_scale p (n-1) in
        fun frame ->
            painter frame;
            rs frame
```

=>

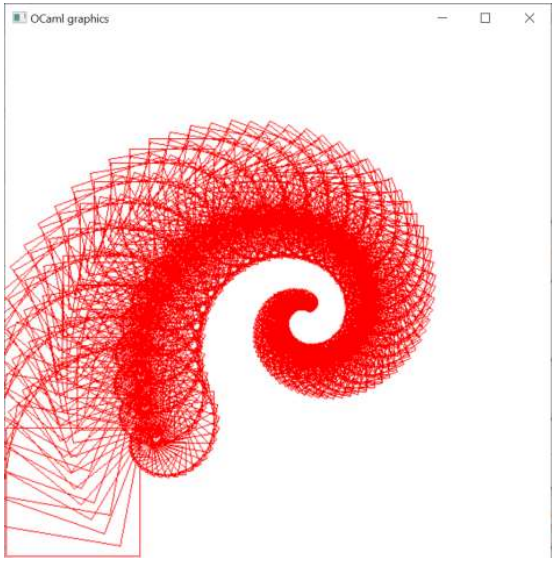


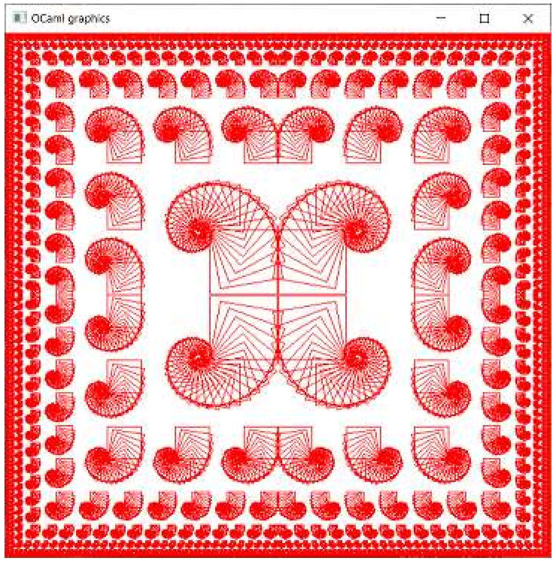


Drawing on a window

```
(*draw--------------------------------------------
*)
let draw painter frame =
    open_graph " 600x600";
    clear_graph ();
    painter frame;
    (*close_graph ();*)
    ()
```

This space is not a mistake





A Picture Language: Overall Program

```
#load "graphics.cma";;
open Graphics;;

(*vector 2d*)
(*frame*)
(*base painter*)
(*simple  transform on painter*)
(*complex transform on painter*)
(*draw*)

let rs = rot_scale (scale 0.5 0.5 base_painter) 50
let p1 = base_painter
let p2 = flip_ver rs
let p3 = flip_hor rs
let p4 = beside rs rs
let p5 = below  rs rs
let p6 = flipped_pairs rs
let p7 = right_split rs 8
let p8 = up_split rs 8
let p9 = corner_split rs 8
let pa = rot_scale (scale 0.5 0.5 rs) 50
let _ = draw p9 frame_g
```

Load Graphics module

After open, you can use lineto instead of Graphics.lineto

Compound Data: Lists

- List
  - Any number of items of the same type
  - Tuple: fixed number of possibly different types
  - E.g.)

```
# [1; 2; 3];;
- : int list = [1; 2; 3]

# ["hello"; "world"];;
- : string list = ["hello"; "world"]

# [1, 2, 3];; (*semicolons vs commas*)
- : (int * int * int) list = [(1, 2, 3)]
```

Compound Data: Lists

- Constructing lists with ::

```
# 1::2::3::[];; (* two list constructors: [] and :: *)
- : int list = [1; 2; 3]

# 1::(2::(3::[]));;
- : int list = [1; 2; 3]

# [1;2;3] @ [4;5;6];; (* list concatenation *)
- : int list = [1; 2; 3; 4; 5; 6]

# [];;
- : 'a list = []
```

Compound Data: Lists

- Use pattern matching to extract components
  - Two list constructors: [ ] and : :

```
let rec sum l =
    match l with
    | [] -> 0
    | hd :: tl -> hd + sum tl
sum [1;2;3];;

-  : int = 6

let rec sum = function
    | [] -> 0
    | hd :: tl -> hd + sum tl
```

function is equivalent to
<param> match <param> with

Compound Data: Lists

- Mapping over list
  - Apply a transform to each element in a list and generate the list of results

```
let rec map f l =
    match l with
    | [] -> []
    | hd :: tl -> (f hd) :: map f tl;;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

let _ = map (fun x -> x * x) [1; 2; 3];;

- : int list = [1; 4; 9]
```

Compound Data: Lists

- Filter
  - Apply a predicate function to each element in a list and generate a filtered list

```
let rec filter f l =
    match l with
    | [] -> []
    | hd :: tl -> if f hd
                  then hd :: filter f tl
                  else filter f tl
let _ = filter ((fun x -> x mod 2 = 0)) [1; 2; 3; 4; 5]

- : int list = [2; 4]
```

Compound Data: Lists

- Function composition by |> operator

```
let sum_of_odd_squares l =
    l |> filter (fun x -> x mod 2 = 1)
      |> map (fun x -> x * x)
      |> sum

let _ = sum_of_odd_squares [1;2;3;4;5;6;7;8;9;10];;
- : int = 165
```

Compound Data: Records

- Records
  - Similar to tuples
  - Individual fields are named

- Defining new data type

```
# type point2d = { x : float; y : float };;
type point2d = { x : float; y : float; }

# let p = { x = 3.; y = -4. };;
val p : point2d = {x = 3.; y = -4.}
```

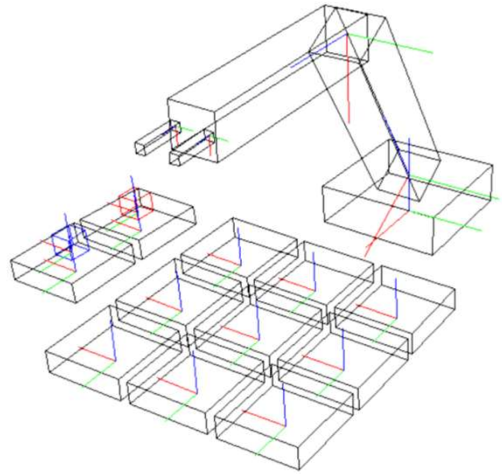
- Accessing data

function parameter

```
let mag1 { x = _x; y = _y } = (*pattern matching*)
    sqrt (_x ** 2. +. _y ** 2.)

let mag2 { x; y } = (*field punning*)
    sqrt (x ** 2. +. y ** 2.)

let mag3 p = (*dot notation*)
    sqrt (p.x ** 2. +. p.y ** 2.)

let mag = mag3

let dist p q = (*distance between p and q*)
    mag { x = p.x -. q.x; y = p.y -. q.y}

let p = { x = 3.; y = -4. }
let q = { x = 4.; y = -5. }

let _= dist p q
- : float = 1.4142135623730951
```

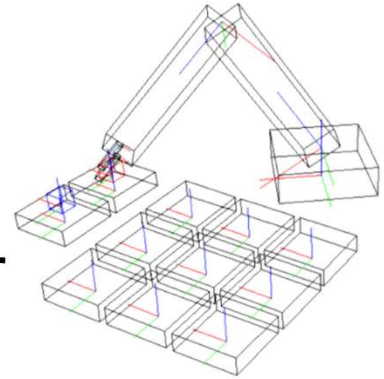
omitting param. names when they are equal to field names

Assignment 3



- Implement a Tic-Tac-Toe game
  - Download robot.zip
  - Implement all TODO parts
  - After finishing the assignment,
    you should be able to play the Tic-Tac-Toe game with the robot
  - Upload basis.ml board.ml, command.ml, drawer.ml, pose.ml, vector.ml in a single zip file to Brightspace
- Due date: 4/4/2024

Abstraction Barriers



Game Plays the game

winner, next_mark, game, ...

Command moves robots

move_to_pose, pick, drop, mark , …

Drawer draws a robot and a board w.r.t. a basis

draw_box, draw_robot, draw_arm1, …

Pose pose of a robot

get_pose, chg_pose, find_pose, …

Basis as a tuple of vectors

scale, translate, rot, v2g, b2g, …

3D vectors as tuples

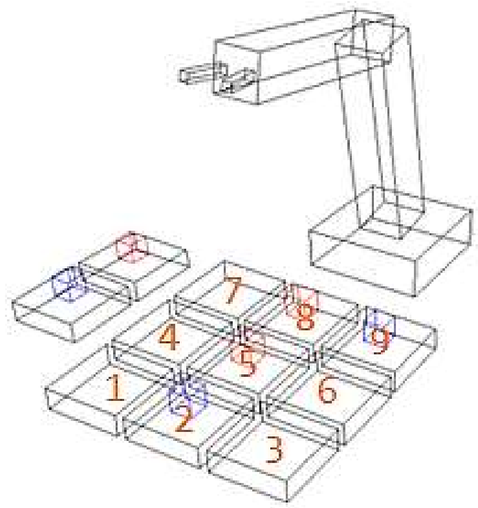
add, sub, prod, smul, …

SUNY Korea

Assignment 3

- To play Tic-Tac-Toe
  - Press the number keys (1 ~ 9) to put a mark at the position
  - Press q to quit

- The robot should mark on the position, where
  - it will win the game if the position is marked by the robot
  - it will lose the game if the position is marked by the other
  - Otherwise, mark any empty position



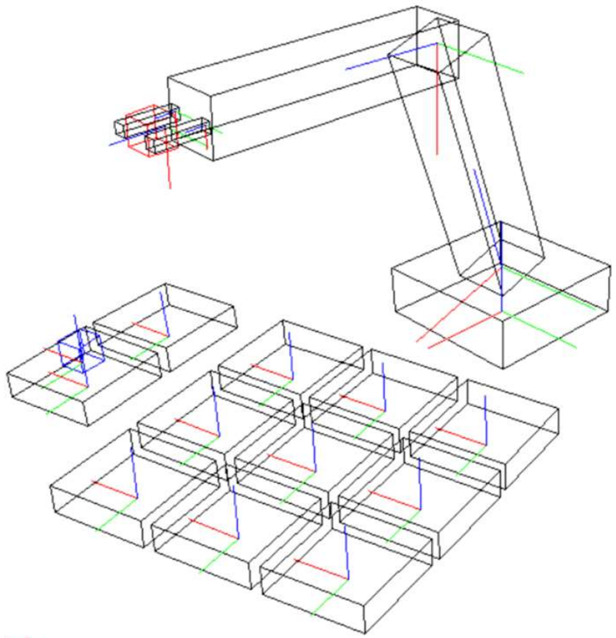
```
(*app.ml*)
…
#use "globals.ml"
#use "vector.ml"
#use "basis.ml"
#use "board.ml"
#use "pose.ml"
#use "drawer.ml"
#use "command.ml"
#use "game.ml"

let app () =
…
    (*camera basis*)
    let b_camera = (b_rotx (-60.) (b_rotz (-210.) gb_basis)) in
    (*initial pose*)
    let ipose = (90., 30., 60., 0., mark_n) in
    (*initial board*)
    let iboard = [ mark_n; mark_n; mark_n;
                   mark_n; mark_n; mark_n;
                   mark_n; mark_n; mark_n;
                   mark_o (*9*); mark_x (*10*)] in

    Graphics.open_graph " 800x800";
    Graphics.auto_synchronize false;
    game b_camera (ipose, iboard) |> print_result;
    Graphics.auto_synchronize true
let _ = app ()
```

Abstraction levels

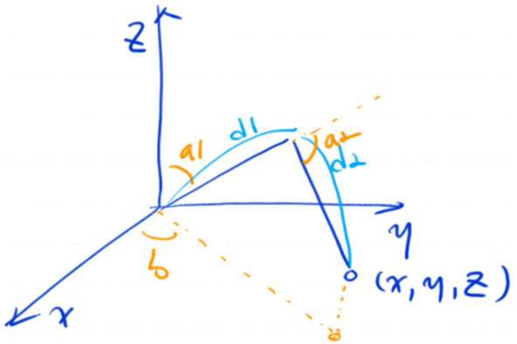
You can test each file by uncommenting test codes

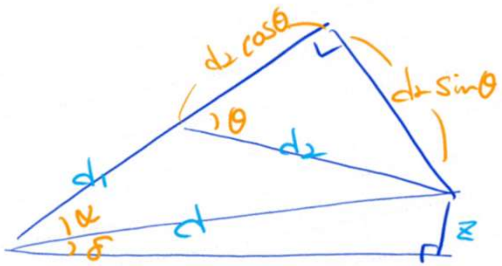
```
(*drawer.ml*)

(*convert b w.r.t. basis to the global coordinate*)
let b2g_basis b basis =
…

let draw_arm1 pose =
    let s     = 0.9 in
    let v_ta2 = (0.0,0.0,0.56) in
    fun basis ->
        let b_a2     = gb_basis  (*b_a2: basis for arm 2*)
                     (*TODO: rotate gb_basis by arm2 angle of pose around y axis*)
                     (*TODO: scale the result by 0.5*)
                     (*TODO: translate the result by v_ta2*)
                     (*TODO: convert the result in basis coord to global coord*)
                     |> b_roty (get_pose pose "arm2")
                     |> b_scale 0.5
                     |> b_translate v_ta2
                     |> fun b -> b2g_basis b basis in

        (*draw arm2 in b_a2 basis*)
        draw_arm2 pose b_a2;

        (*draw arm1*)
        draw_box (0.12/.s) (0.12/.s) (0.5/.s) Graphics.black basis
```

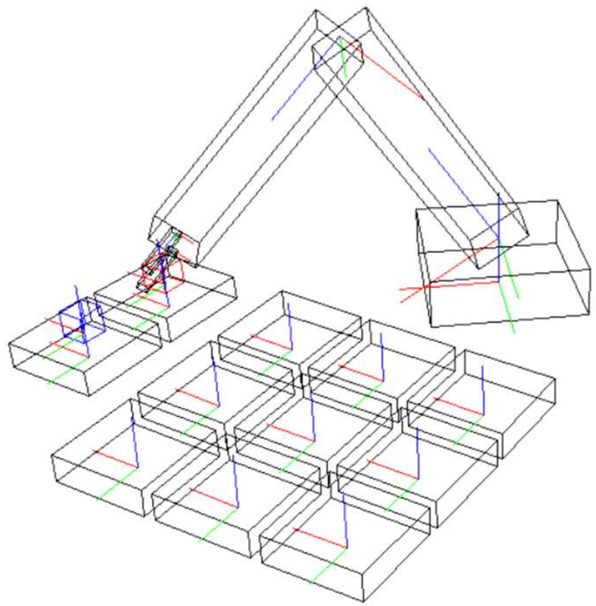
These are not in your assignment file



```
(*pose.ml*)

type pose = float * float * float * float * float;;

(*find the angle of joints to get to x y z*)
let find_pose (x, y, z) =
    fun f m ->
        (*TODO: find b, a1, and a2 and return the pose (b, a1, a2, f, m)
                 b: angle (deg) of base measured from x axis (use atan2),
                a1: angle (deg) of arm1 measured from z axis
                a2: angle (deg) of arm2 measured from arm1 … *)
```







$$d = \sqrt{x^2 + y^2 + z^2}$$

$$(d_1 + d_2 \cos\theta)^2 + (d_2 \sin\theta)^2 = d^2$$

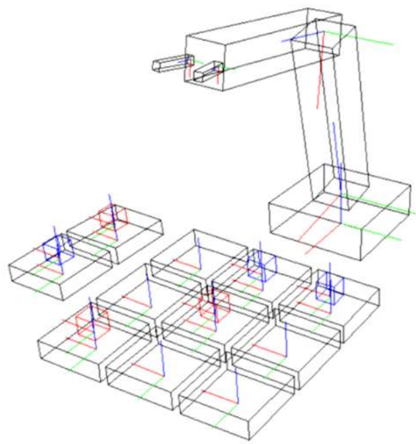
$$\sin\alpha = d_2 \sin\theta / d$$

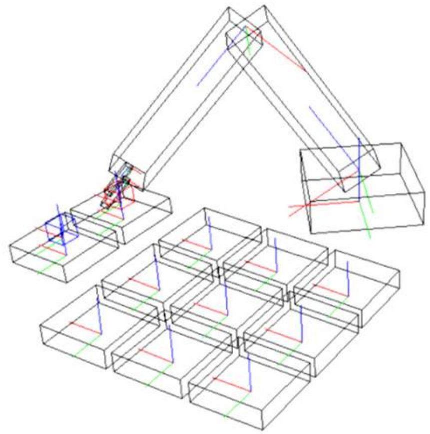
$$\sin\delta = z / d$$

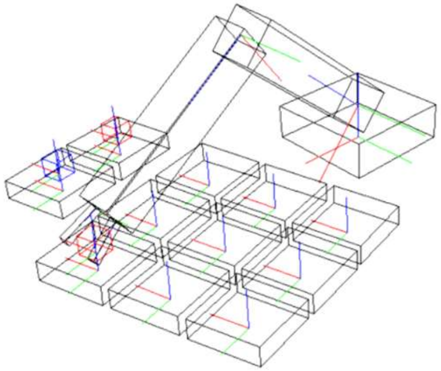
$$\tan b = \frac{y}{x}$$

SUNY Korea

```ocaml
(*command.ml*)

(*move from pose to target_pose*)
let moveto_pose b_camera (pose, board) target_pose =
    let db  = (get_pose target_pose "base")   -. (get_pose pose "base") in
    let da1 = (get_pose target_pose "arm1")   -. (get_pose pose "arm1") in
    let da2 = (get_pose target_pose "arm2")   -. (get_pose pose "arm2") in
    let df  = (get_pose target_pose "finger") -. (get_pose pose "finger") in

    (*move the joint <ang> angle in <step> steps
      e.g. rotate arm1 30 deg in 5 steps
           => rotate arm1 5 times 6 deg each
    *)
    let rot_joint pose joint ang step =
        (*TODO: implement this method
            - on each step, draw the robot and the board
            - wait for 50ms by calling Thread.delay 0.05
            - after rotating step times, return the final pose
        *)


    (*move the joints in base, arm1, arm2, and finger order*)
    let p    = pose
            |> fun p -> rot_joint p "base" db  5
            |> fun p -> rot_joint p "arm1" da1 5
            |> fun p -> rot_joint p "arm2" da2 5
            |> fun p -> rot_joint p "finger" df 3 in
    (p, board)
```







```
(*command.ml*)

(*put mark at dst*)
let mark b_camera (pose, board) mrk dst =
    let src = if mrk = mark_o then 9 else 10 in
    let f = get_pose pose "finger" in
    let m = get_pose pose "mark" in

    (*TODO: 1) find b, a1, and a2 for dst_pose and src_pose
               using find_pose, mark_pos then
            2) pass two params for the fun returned by find_pose
    *)
    let dst_pose = in (*robot's pose for the dst-th mark with finger is f, mark is mrk*)
    let src_pose = in (*robot's pose for the src-th mark with finger is 0, mark is m*)

    (*moveto_pose with the first param applied*)
    let mvp = moveto_pose b_camera in

    (*TODO: 1. move to pose src_pose (use mvp)
            2. pick the mark at src   (use pick)
            3. lift                   (use mvp and lift_pose)
            4. move to pose dst_pose (use mvp)
            5. drop the mark at dst   (use drop)
            6. lift                   (use mvp and lift_pose)
            7. return the resulting pose and the board*)
```